CS152 Discussion Section

Branch Predictions and VLIW

Mar 11-15
Spring 2024

Agenda

- VLIW
  - Software pipelining
- Branch prediction
  - Branch history table
  - Branch target buffer
- Lab 3 overview

Administrivia

- HW3 due March 18
- Lab 3 due March 20

VLIW

VLIW

- Multiple parallel operations packed into a single instruction
  - Each operation slot is dedicated to a fixed function
  - Resembles horizontal microcode engine
- Classic VLIW machines have exposed pipelines with no hardware interlocks
  - Exploit ILP without complexity of OoO superscalar control logic by shifting burden to compiler
  - Rely entirely on the compiler to explicitly schedule operations around data hazards
  - Requires latencies to be statically known
- Static scheduling vs dynamic scheduling
- Software techniques
  - Loop unrolling
  - Software pipelining
  - Trace scheduling

More information in online Appendix H of H&P textbook (6th edition)

VLIW: Dot Product (Q1)

```
for (i = 0; i < N; i++)
     C += A[i] * B[i]
```

```
loop:
     fld f1, 0(x1)
     fld f2, 0(x2)
     fmul f3, f1, f2
     fadd f0, f0, f3
     addi x1, x1, 8
     addi x2, x2, 8
     bne x1, x3, loop
```

All functional units fully pipelined

- 1-cycle integer ALU
- 2-cycle load/store unit
- 3-cycle floating-point adder
- 4-cycle floating-point multiplier

Label	ALU	MEM	FADD	FMUL
loop:				

N-cycle latency means result usable in N cycles (i.e. load in cycle 0 is usable in cycle 2)

VLIW: Dot Product (Q1)

```
for (i = 0; i < N; i++)
     C += A[i] * B[i]

loop:
     fld f1, 0(x1)
     fld f2, 0(x2)
     fmul f3, f1, f2
     fadd f0, f0, f3
     addi x1, x1, 8
     addi x2, x2, 8
     bne x1, x3, loop
```

All functional units fully pipelined

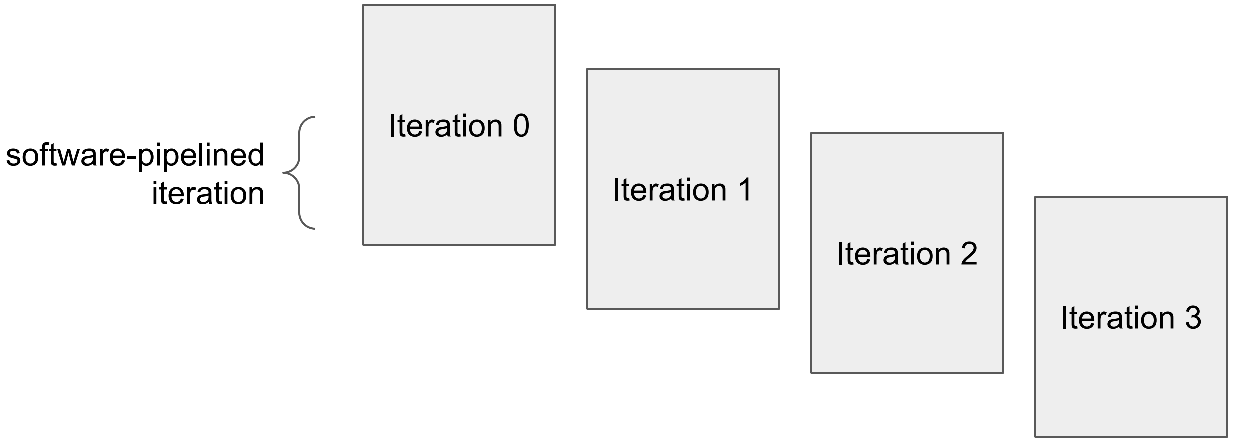
- 1-cycle integer ALU
- 2-cycle load/store unit
- 3-cycle floating-point adder
- 4-cycle floating-point multiplier

Label	ALU	MEM	FADD	FMUL
loop:	addi x1	fld f1		
	addi x2	fld f2		
				fmul f3
			fadd f0	
	bne loop			

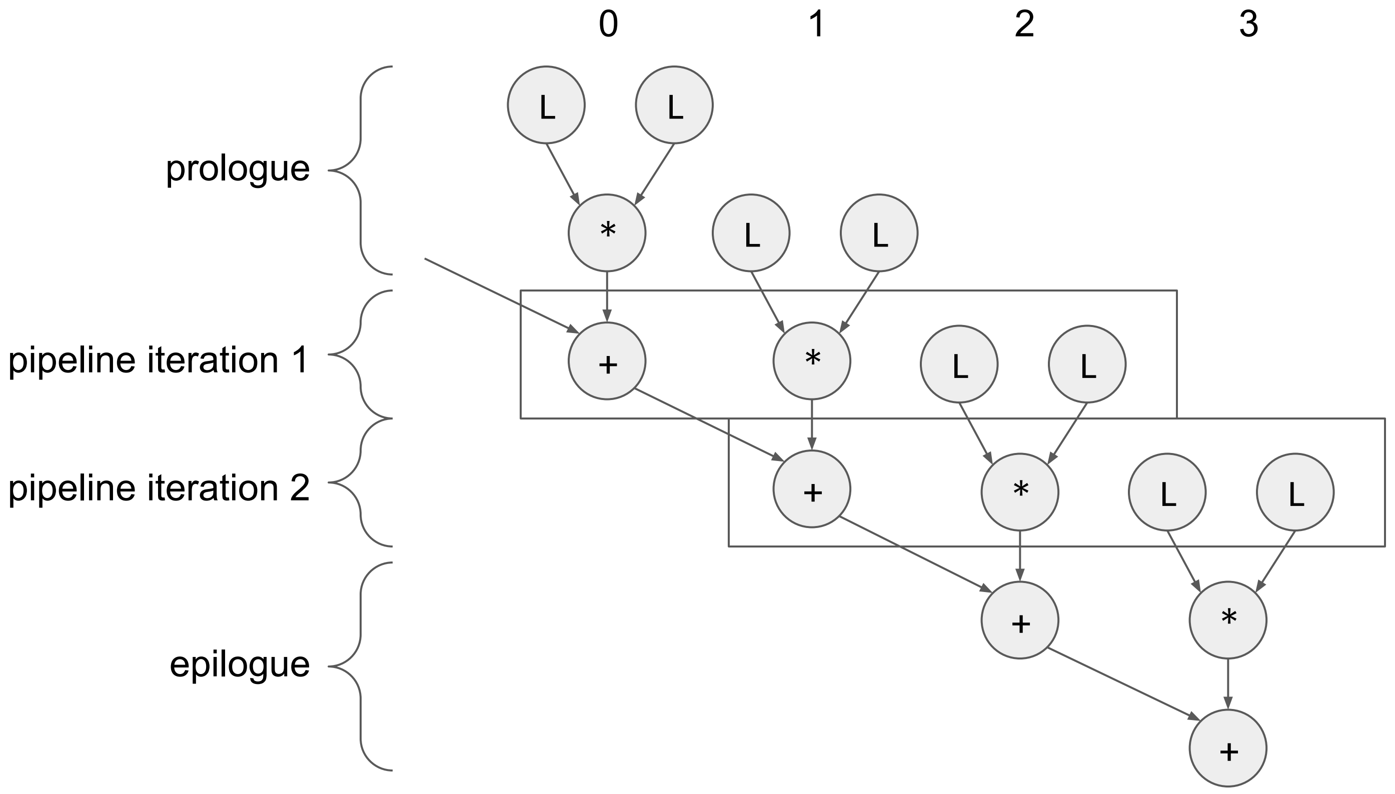
gotta go faster

Software Pipelining

- Classic software technique for restructuring loops
- Interleaves instructions from different iterations
  - Works with and without loop unrolling
- Dependent operations are separated by one loop body, decreasing stalls
- Start-up (prologue) and wind-down (epilogue) code needed before/after loop



Software Pipelining: Dot Product



Software Pipelining: Scalar Version

```
loop:
     fld f1, 0(x1)
     fld f2, 0(x2)
     fmul f3, f1, f2
     fadd f0, f0, f3
     addi x1, x1, 8
     addi x2, x2, 8
     bne x1, x3, loop
```

- Iteration *i*:

```
fadd f0, f0, f3
```

- Iteration *i+1*:

```
fmul f3, f1, f2
```

- Iteration *i+2*:

```
fld f1, 0(x1)
fld f2, 0(x2)
addi x1, x1, 8
addi x2, x2, 8
```

```
prologue:
     ...
loop:
     fadd f0, f0, f3
     fmul f3, f1, f2
     fld f1, 0(x1)
     fld f2, 0(x2)
     addi x1, x1, 8
     addi x2, x2, 8
     bne x1, x3, loop
epilogue:
     ...
```

Software Pipelining: VLIW

```
loop:
     fld f1, 0(x1)
     fld f2, 0(x2)
     fmul f3, f1, f2
     fadd f0, f0, f3
     addi x1, x1, 8
     addi x2, x2, 8
     bne x1, x3, loop
```

- 1-cycle integer ALU
- 2-cycle load/store unit
- 3-cycle floating-point adder
- 4-cycle floating-point multiplier

Assume perfect branch prediction

Label	ALU	MEM	FADD	FMUL

Software Pipelining: VLIW

```
loop:
     fld f1, 0(x1)
     fld f2, 0(x2)
     fmul f3, f1, f2
     fadd f0, f0, f3
     addi x1, x1, 8
     addi x2, x2, 8
     bne x1, x3, loop
```

- 1-cycle integer ALU
- 2-cycle load/store unit
- 3-cycle floating-point adder
- 4-cycle floating-point multiplier

Assume perfect branch prediction

Label	ALU	MEM	FADD	FMUL
	addi x1	fld f1		
	addi x2	fld f2		
	addi x1	fld f1		fmul f3
	addi x2	fld f2		
loop:	addi x1	fld f1		fmul f3
	addi x2	fld f2	fadd f0	
	bne loop			
				fmul f3
			fadd f0	
			fadd f0	

Software Pipelining: VLIW

```
loop:
    fld f1, 0(x1)
    fld f2, 0(x2)
    fmul f3, f1, f2
    fadd f0, f0, f3
    addi x1, x1, 8
    addi x2, x2, 8
    bne x1, x3, loop
```

- 1-cycle integer ALU
- 2-cycle load/store unit
- 3-cycle floating-point adder
- 4-cycle floating-point multiplier

Assume perfect branch prediction

Label	ALU	MEM	FADD	FMUL
	Addi x1	Fld f1		
	Addi x2	Fld f2		
	Addi x1	Fld f1		
	Addi x2	Fld f2		Fmul f3
	Addi x1	Fld f1		
	Addi x2	Fld f2		Fmul f3
loop	Addi x1	Fld f1		
	Addi x2	Fld f2	Fadd f0	Fmul f3
	bne			
			Fadd f0	Fmul f3
			Fadd f0	
			Fadd f0	

gotta go faster..er

Software Pipelining + Unrolling

- Unroll loop 4 times
  - Accumulate 4 partial sums in parallel (f0, f1, f2, f3)
  - Reduce at end (not shown)
- Assume VLIW machine has second load/store unit
  - Dot product has 1:1 ratio between loads and FLOPs
- Yields optimal throughput of 2 FLOPs/cycle

```
loop:
    fld   f4,    0(x1)
    fld   f5,    0(x2)
    fld   f6,    8(x1)
    fld   f7,    8(x2)
    fld   f8,   16(x1)
    fld   f9,   16(x2)
    fld  f10,   24(x1)
    fld  f11,   24(x2)
    fmul f12,  f4,  f5
    fmul f13,  f6,  f7
    fmul f14,  f8,  f9
    fmul f15, f10, f11
    fadd  f0,  f0, f12
    fadd  f1,  f1, f13
    fadd  f2,  f2, f14
    fadd  f3,  f3, f15
    addi x1, x1, 32
    addi x2, x2, 32
    bne x1, x3, loop
```

```
loop:
  fld   f4,    0(x1)
  fld   f5,    0(x2)
  fld   f6,    8(x1)
  fld   f7,    8(x2)
  fld   f8,   16(x1)
  fld   f9,   16(x2)
  fld  f10,   24(x1)
  fld  f11,   24(x2)
  fmul f12,  f4,  f5
  fmul f13,  f6,  f7
  fmul f14,  f8,  f9
  fmul f15, f10, f11
  fadd  f0,  f0, f12
  fadd  f1,  f1, f13
  fadd  f2,  f2, f14
  fadd  f3,  f3, f15
  addi  x1,  x1, 32
  addi  x2,  x2, 32
  bne   x1,  x3, loop
```

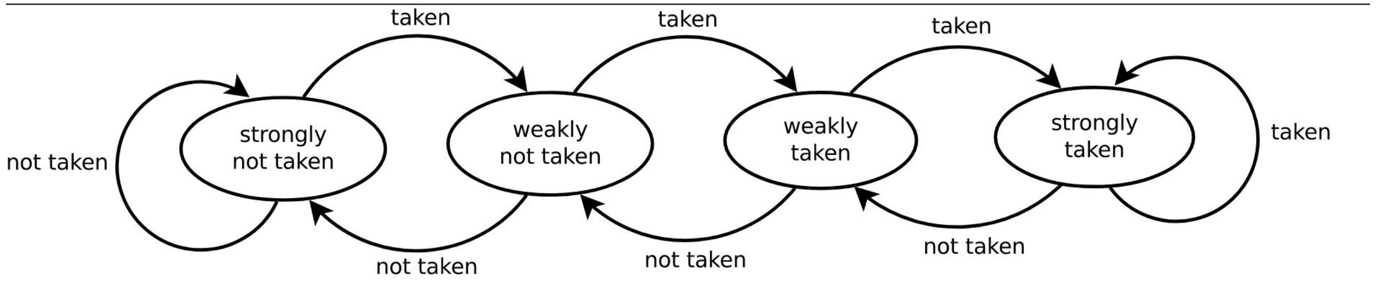
Label	INT	MEM0	MEM1	FADD	FMUL

```
loop:
  fld   f4,    0(x1)
  fld   f5,    0(x2)
  fld   f6,    8(x1)
  fld   f7,    8(x2)
  fld   f8,   16(x1)
  fld   f9,   16(x2)
  fld  f10,   24(x1)
  fld  f11,   24(x2)
  fmul f12,  f4,  f5
  fmul f13,  f6,  f7
  fmul f14,  f8,  f9
  fmul f15, f10, f11
  fadd  f0,  f0, f12
  fadd  f1,  f1, f13
  fadd  f2,  f2, f14
  fadd  f3,  f3, f15
  addi  x1,  x1, 32
  addi  x2,  x2, 32
  bne   x1,  x3, loop
```

Label	INT	MEM0	MEM1	FADD	FMUL
	addi x1,x1,32	fld f4,0(x1)	fld f5,0(x2)		
	addi x2,x2,32	fld f6,-24(x1)	fld f7,8(x2)		
		fld f8,-16(x1)	fld f9,-16(x2)		
		fld f10,-8(x1)	fld f11,-8(x2)		
	addi x1,x1,32	fld f4,0(x1)	fld f5,0(x2)		fmul f12,f4,f5
	addi x2,x2,32	fld f6,-24(x1)	fld f7,8(x2)		fmul f13,f6,f7
		fld f8,-16(x1)	fld f9,-16(x2)		fmul f14,f8,f9
		fld f10,-8(x1)	fld f11,-8(x2)		fmul f15,f10,f11
loop:	addi x1,x1,32	fld f4,0(x1)	fld f5,0(x2)	fadd f0,f0,f12	fmul f12,f4,f5
	addi x2,x2,32	fld f6,-24(x1)	fld f7,8(x2)	fadd f1,f1,f13	fmul f13,f6,f7
		fld f8,-16(x1)	fld f9,-16(x2)	fadd f2,f2,f14	fmul f14,f8,f9
	bne x1,x3,loop	fld f10,-8(x1)	fld f11,-8(x2)	fadd f3,f3,f15	fmul f15,f10,f11
				fadd f0,f0,f12	fmul f12,f4,f5
				fadd f1,f1,f13	fmul f13,f6,f7
				fadd f2,f2,f14	fmul f14,f8,f9
				fadd f3,f3,f15	fmul f15,f10,f11
				fadd f0,f0,f12	
				fadd f1,f1,f13	
				fadd f2,f2,f14	

Branch Prediction

Bimodal Counters



- 2-bits of state. Upper bit encodes direction
- Increment saturating counter for taken, decrement for not-taken

```
for (int i = 0; i < 4; i++)    // BRANCH_A (Taken means stay in the loop)
  for (int j = 0; j < 4; j++)  // BRANCH_B (Taken means stay in the loop)
    if (j % 2)                 // BRANCH_C (Taken means perform the body of the if)
      sum += i
```

Assume a BHT indexed by PC, the PCs of the branches do not alias into the same entry

- What is the prediction accuracy for each branch with 1-bit counters?
- What is the prediction accuracy for each branch with 2-bit counters?

1-bit Bimodal Counters (Q2.1)

Assume counters are initialized to 0 (not taken)

```
for (int i = 0; i < 4; i++)              // BRANCH_A
     for (int j = 0; j < 4; j++)         // BRANCH_B
          if (j % 2 == 0)                // BRANCH_C
               sum += i
```

Branch A:

	i=0	i=1	i=2	i=3	i=4
Counter	0				
Predictio n					
Actual					

Branch B:

	j=0	j=1	j=2	j=3	j=4	j=0	j=1	...
Counter	0							...
Prediction								...
Actual								...

Branch C:

	j=0	j=1	j=2	j=3	j=0	j=1	...
Counter	0						...
Prediction							...
Actual							...

1-bit Bimodal Counters (Q2.1)

Assume counters are initialized to 0 (not taken)

```
for (int i = 0; i < 4; i++)           // BRANCH_A
    for (int j = 0; j < 4; j++)       // BRANCH_B
        if (j % 2 == 0)               // BRANCH_C
            sum += i
```

Branch A:

	i=0	i=1	i=2	i=3	i=4
Counter	0	1	1	1	1
Predictio n	0	1	1	1	1
Actual	1	1	1	1	0

Branch B:

	j=0	j=1	j=2	j=3	j=4	j=0	j=1	...
Counter	0	1	1	1	1	0	1	...
Prediction	0	1	1	1	1	0	1	...
Actual	1	1	1	1	0	1	1	...

Branch C:

	j=0	j=1	j=2	j=3	j=0	j=1	...
Counter	0	1	0	1	0	1	...
Prediction	0	1	0	1	0	1	...
Actual	1	0	1	0	1	0	...

2-bit Bimodal Counters (Q2.2)

Assume counters are initialized to 00 (strongly not taken)

```
for (int i = 0; i < 4; i++)           // BRANCH_A
      for (int j = 0; j < 4; j++)      // BRANCH_B
            if (j % 2 == 0)                 // BRANCH_C
                  sum += i
```

Branch A:

	i=0	i=1	i=2	i=3	i=4
Counter	00				
Predictio n					
Actual					

Branch B:

	j=0	j=1	j=2	j=3	j=4	j=0	j=1	...
Counter	00	01						...
Prediction								...
Actual								...

Branch C:

	j=0	j=1	j=2	j=3	j=0	j=1	...
Counter	00						...
Prediction							...
Actual							...

2-bit Bimodal Counters (Q2.2)

Assume counters are initialized to 00 (strongly not taken)

```
for (int i = 0; i < 4; i++)            // BRANCH_A
      for (int j = 0; j < 4; j++)      // BRANCH_B
            if (j % 2 == 0)                  // BRANCH_C
                  sum += i
```

Branch A:

	i=0	i=1	i=2	i=3	i=4
Counter	00	01	10	11	11
Prediction	0	0	1	1	1
Actual	1	1	1	1	0

Branch B:

	j=0	j=1	j=2	j=3	j=4	j=0	j=1	...
Counter	00	01	10	11	11	10	11	...
Prediction	0	0	1	1	1	1	1	...
Actual	1	1	1	1	0	1	1	...

Branch C:

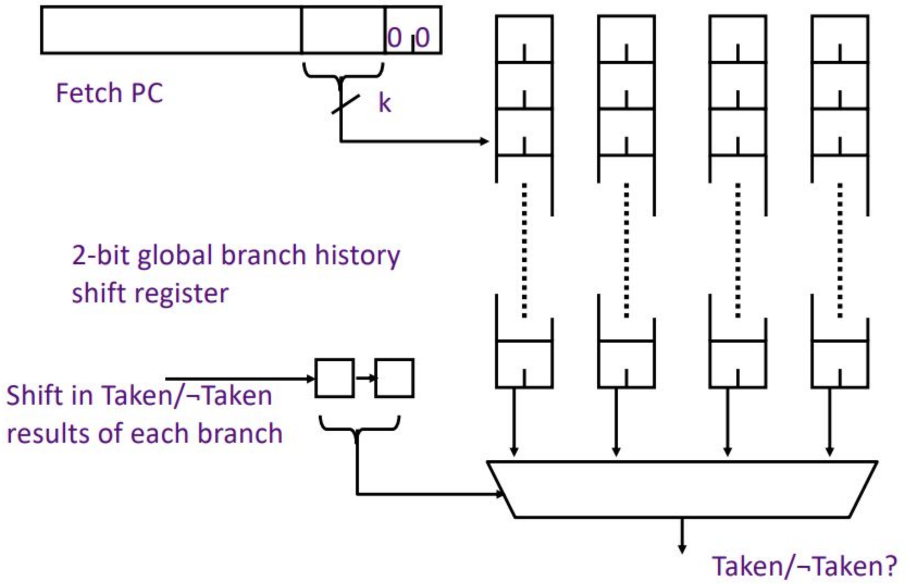
	j=0	j=1	j=2	j=3	j=0	j=1	...
Counter	00	01	00	01	00	01	...
Prediction	0	0	0	0	0	0	...
Actual	1	0	1	0	1	0	...

Two-level Predictors

Idea: Dynamic behavior of past branches can be used to predict future branches

Branch history: bit-vector of T/NT (1/0) representing dynamic branch history

Combine branch history with PC to select a counter


0 0
Fetch PC
k
2-bit global branch history shift register
Shift in Taken/¬Taken results of each branch
Taken/¬Taken?

Two-level Predictors

```
for (int i = 0; i < 4; i++)    // BRANCH_A
  for (int j = 0; j < 4; j++)  // BRANCH_B
    if (j % 2)                 // BRANCH_C
      sum += i
```

Consider a two-level predictor, where a branch history selects a prediction from multiple PC-indexed counter tables.

Could you design this system to perfectly predict every branch?

- If using local histories, how many local history bits?
- If using global histories, how many bits of global bits?

	Predictor policy based on local history	Predictor policy based on global history
BRANCH_A		
BRANCH_B		
BRANCH_C		

Two-level Predictors

```
for (int i = 0; i < 4; i++)     // BRANCH_A
  for (int j = 0; j < 4; j++)   // BRANCH_B
    if (j % 2)                  // BRANCH_C
      sum += i
```

Consider a two-level predictor, where a branch history selects a prediction from multiple PC-indexed counter tables.

Could you design this system to perfectly predict every branch?

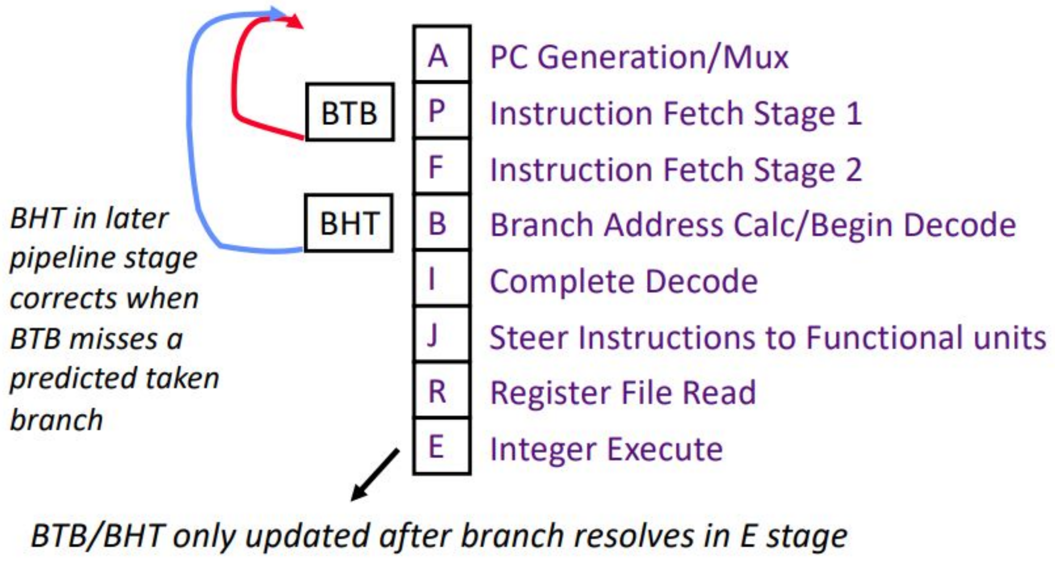
- If using local histories, how many local history bits?
- If using global histories, how many bits of global bits?

	Predictor policy based on local history	Predictor policy based on global history
BRANCH_A	4 (last 4 predictions can determine 5th)	10 * 4
BRANCH_B	4	8
BRANCH_C	1 (flip prior prediction bit)	4 (longest distance to last prediction)

A BC BC BC BC B | A BC BC BC BC B | A BC BC BC BC B | A BC BC BC BC B | A …

Branch Target Buffers

- Branch history tables provide a direction, but what about the target?
- May want to redirect branches even before instruction bits are available
- Return-address-stack: Push RA on function calls, pop to predict target of a ret


BTB
BHT
BHT in later pipeline stage corrects when BTB misses a predicted taken branch
A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute
BTB/BHT only updated after branch resolves in E stage

Branch Predictor Structures

C code	RISCV asm	What structures can and should learn the behavior of this branch?
`for (i=0; i<N; i++)`	`bne t0, a0, loop`	
`if (err) goto ERROR;`	`bnez t0, error`	
`goto LABEL;`	`j LABEL`	
`return;`	`ret (jalr ra)`	
`switch (x)`	`jalr x0, t0`	

Branch Predictor Structures

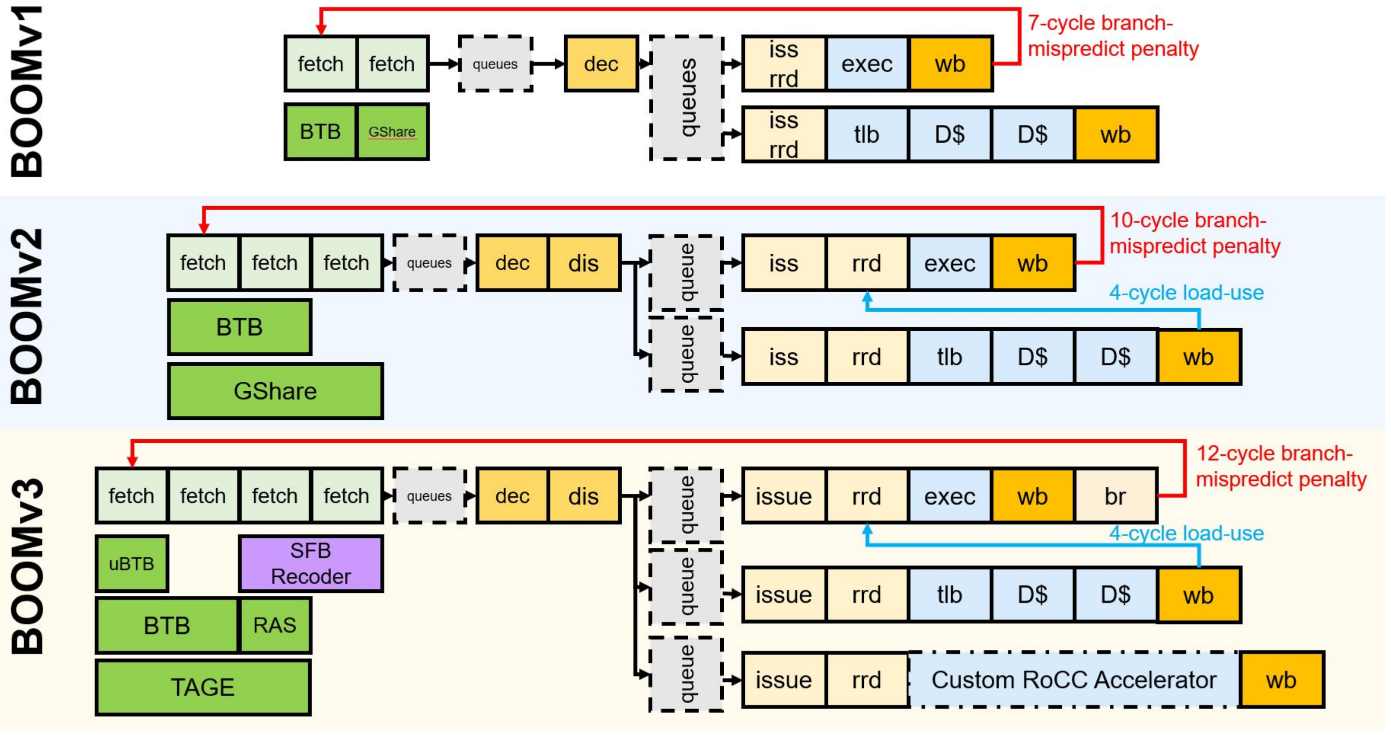
C code	RISCV asm	What structures can and should learn the behavior of this branch?
`for (i=0; i<N; i++)`	`bne t0, a0, loop`	BHT
`if (err) goto ERROR;`	`bnez t0, error`	BHT
`goto LABEL;`	`j LABEL`	BTB
`return;`	`ret (jalr ra)`	RAS
`switch (x)`	`jalr x0, t0`	BTB/Unpredictable

Lab 3 Overview

BOOM: Berkeley Out-of-Order Machine

- Open-source, synthesizable, out-of-order superscalar RISC-V core
- Heavily inspired by the MIPS R10000 and Alpha 21264
- Unified physical register file with explicit renaming
- Split ROB / issue window design
- Extensively parameterized:
    - Fetch and issue widths, ROB size, LSU size
    - Functional unit mix, latencies
    - Issue scheduler
    - Composable branch predictors, RAS size, BTB size
    - Commit map table (R10k rollback vs Alpha 21264 single-cycle flush)
    - Maximum in-flight branches

BOOM: Berkeley Out-of-Order Machine



BOOMv1
fetch fetch queues dec queues
iss rrd exec wb
iss rrd tlb D$ D$ wb
BTB GShare
7-cycle branch-mispredict penalty
BOOMv2
fetch fetch fetch queues dec dis queue queue
iss rrd exec wb
iss rrd tlb D$ D$ wb
BTB
GShare
10-cycle branch-mispredict penalty
4-cycle load-use
BOOMv3
fetch fetch fetch fetch queues dec dis queue queue queue
issue rrd exec wb br
issue rrd tlb D$ D$ wb
issue rrd Custom RoCC Accelerator wb
uBTB
SFB Recoder
BTB RAS
TAGE
12-cycle branch-mispredict penalty
4-cycle load-use

Open-Ended (1): Branch predictor design [Recommended]

- Implement a branch predictor in C++ that integrates with BOOM
- Objective is to beat baseline implementation of BHT
    - Also compare against BOOM’s fine-tuned HW branch predictors

Open-Ended (2): Spectre attacks

- **Spectre/Meltdown**: Microarchitectural side-channel attacks that exploit branch prediction, speculative execution, and cache timing to bypass security mechanisms
- Objective is to recreate Spectre attacks on BOOM
- Attack scenario
  - Vulnerable Spectre gadget present in supervisor syscall code
  - Write user program to infer secret data from protected kernel memory using branch predictor mis-training and cache side effects

FEEDBACK!

https://tinyurl.com/152feedback